Dijkstra's Algorithm: single source shortest paths

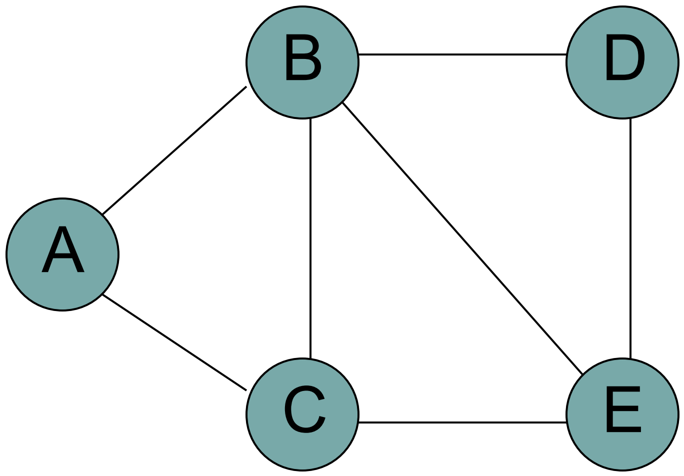
David Kauchak

cs62

Spring 2011

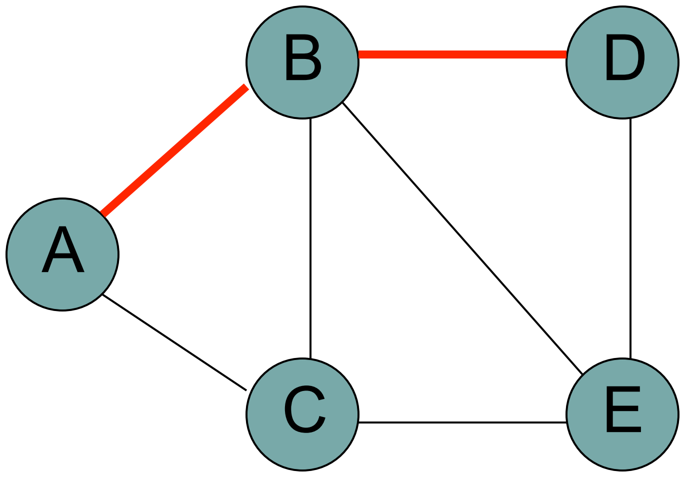
Shortest paths

What is the shortest path from a to d?



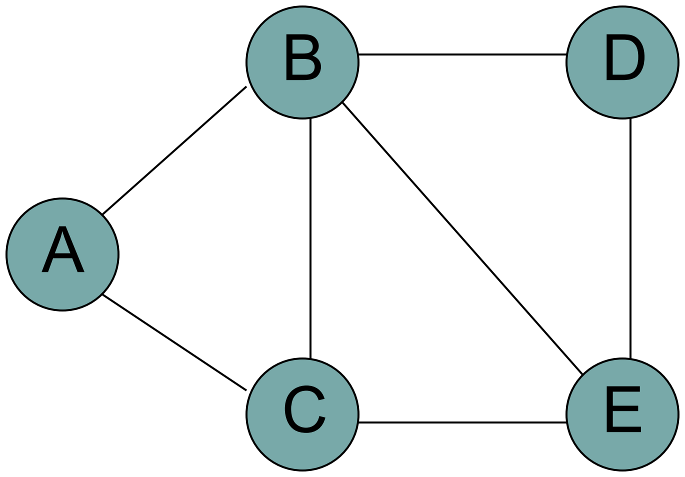
Shortest paths

How can we find this?



Shortest paths

BFS



How does BFS work?

Breadth first search

BFS

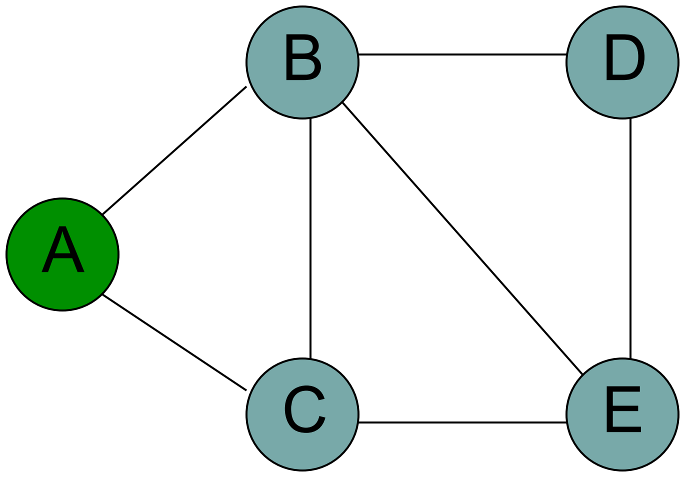
```
enqueue start;

while (queue not empty) {
   dequeue v;

   if (v is not visited) {
     visit v;
     enqueue all of v's neighbors;
   }
}
```

Looks a lot like DFS

How are they different?

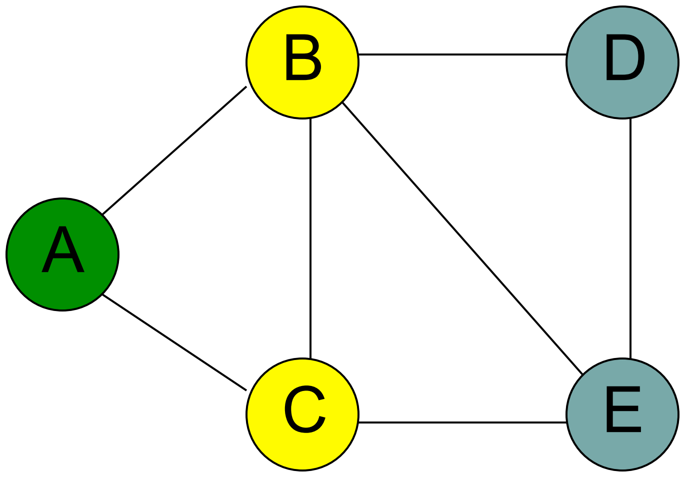
Shortest paths

BFS



Shortest paths

BFS



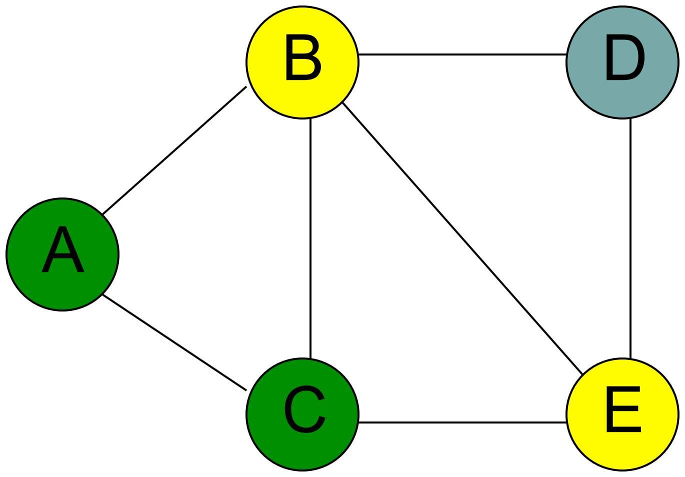
queue

C

B

Shortest paths

BFS



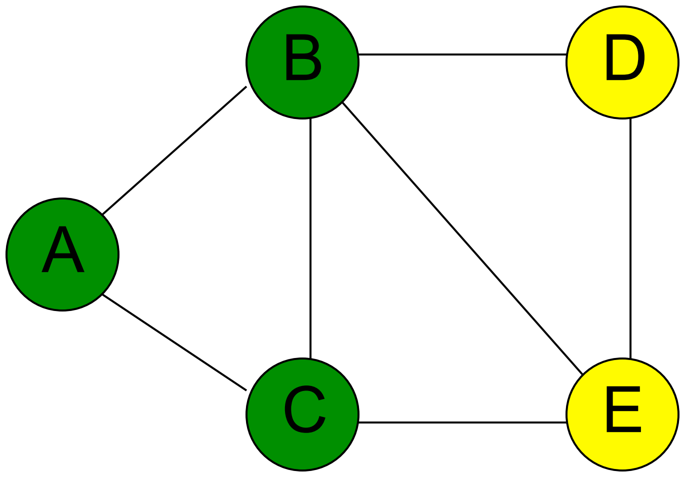
queue

B

E

Shortest paths

BFS



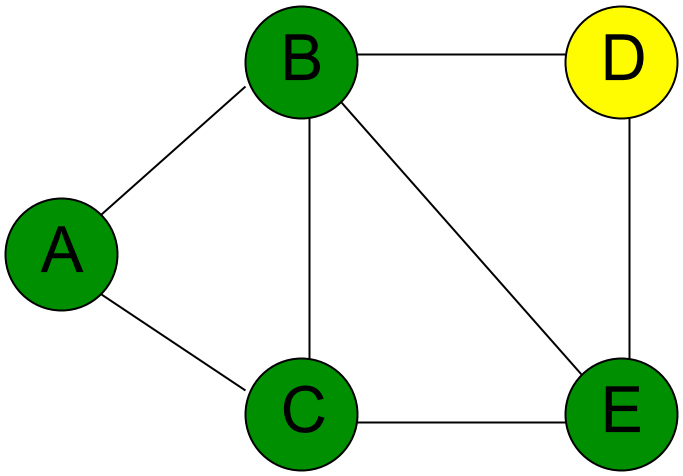
queue

E

D

Shortest paths

BFS

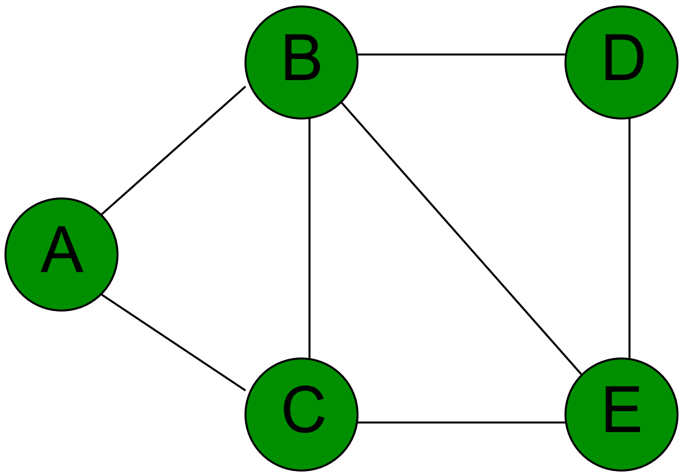


queue

D

Shortest paths

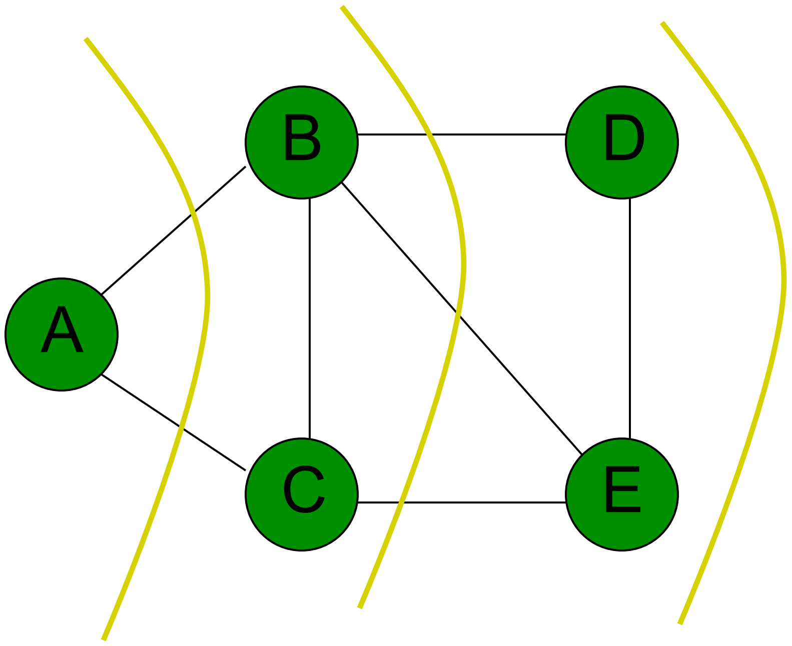
BFS



queue

Shortest paths

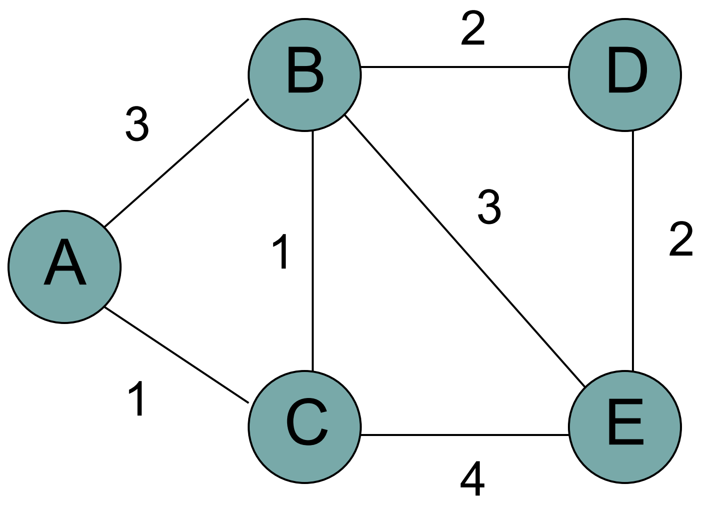
BFS



queue

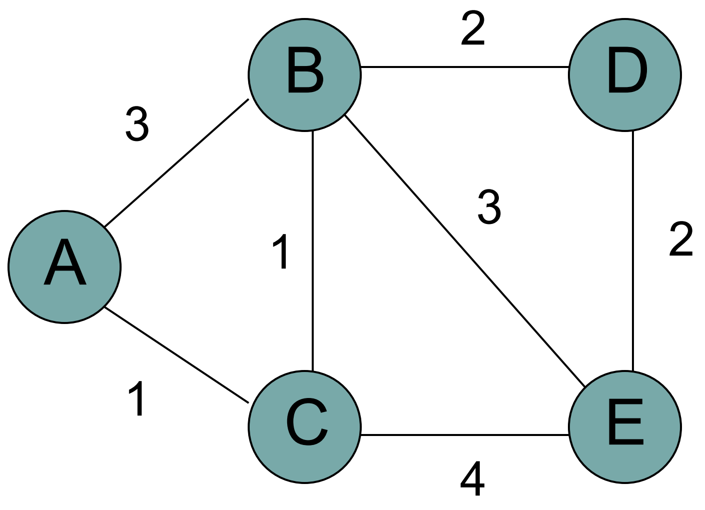
Shortest paths

What is the shortest path from a to d?



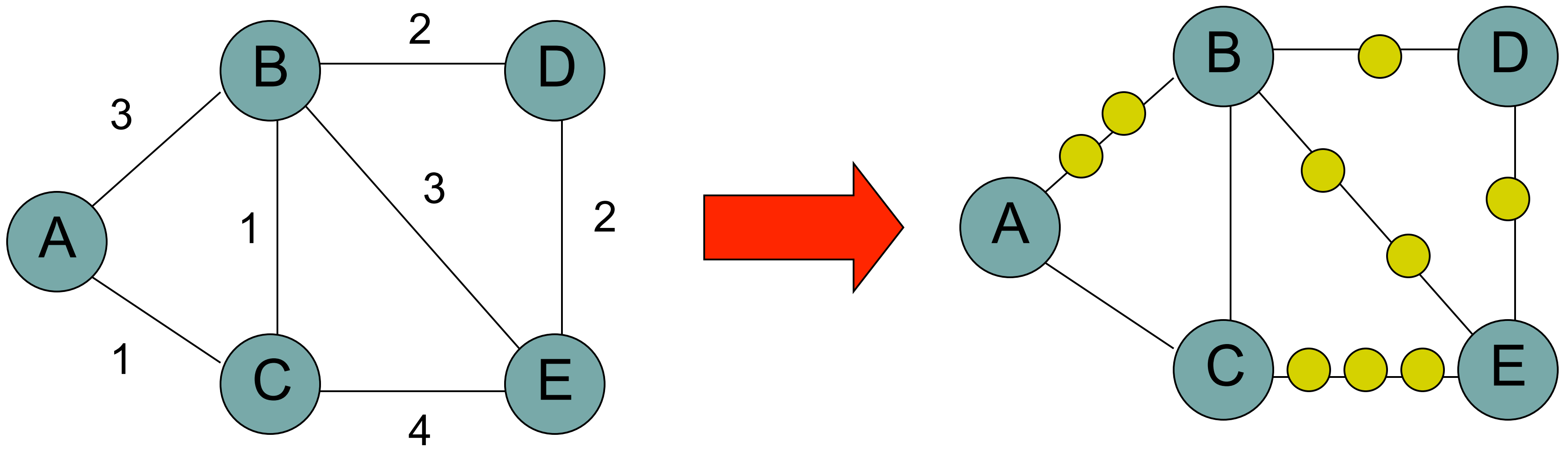
Shortest paths

Can we modify the graph to still use BFS?



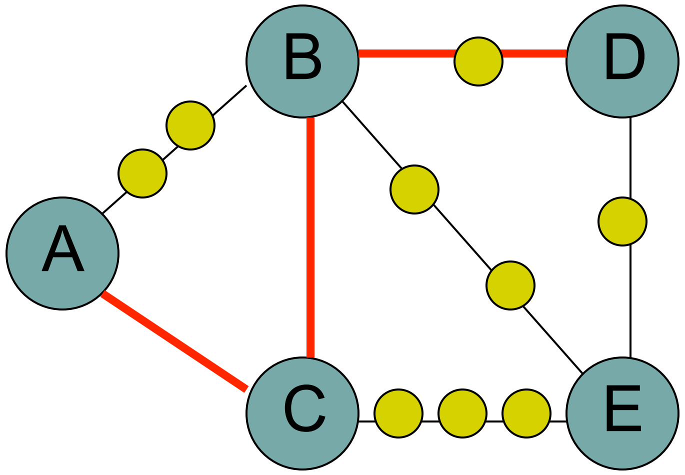
Shortest paths

We can still use BFS



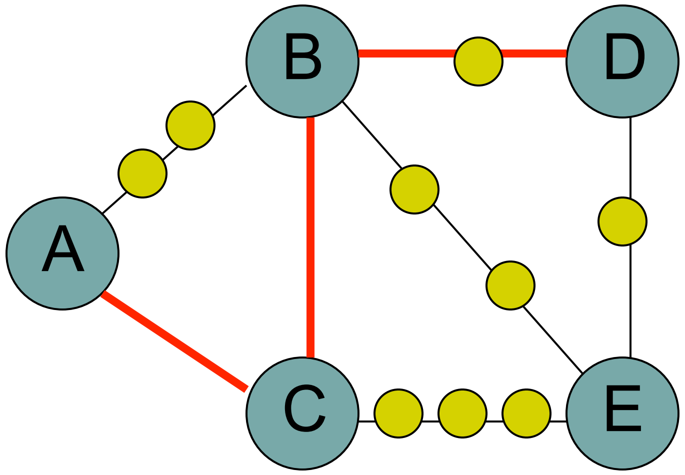
Shortest paths

We can still use BFS



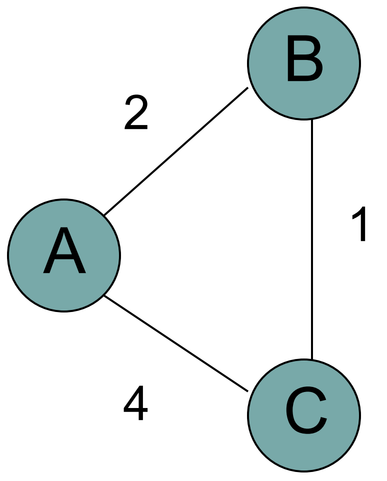
Shortest paths

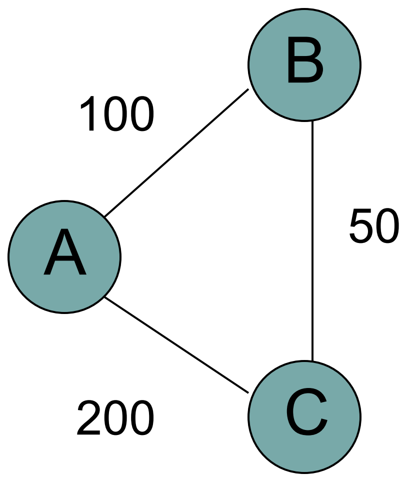
What is the problem?



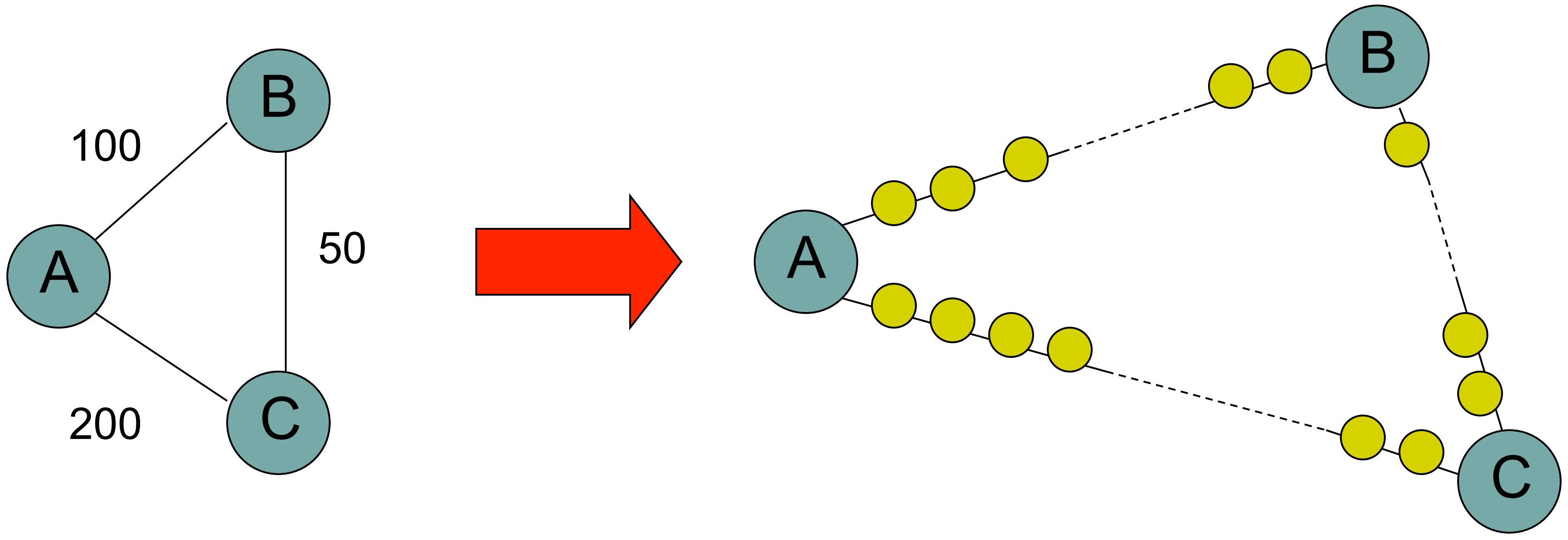
Shortest paths

Running time is dependent on the weights

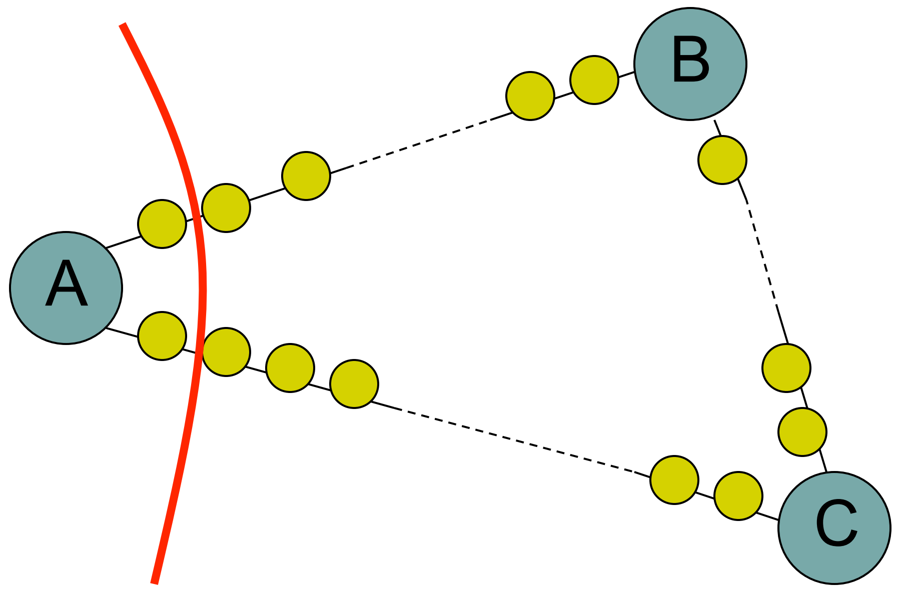




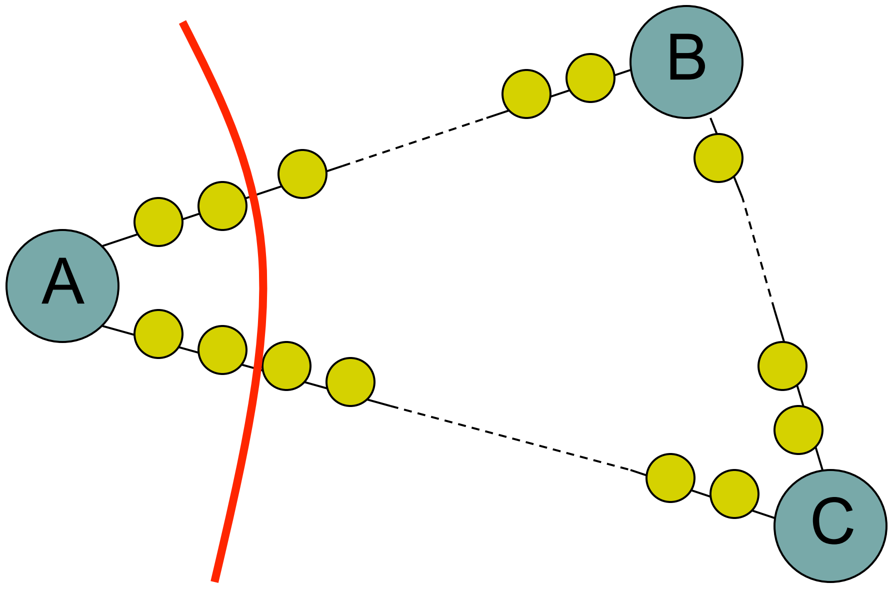
Shortest paths



Shortest paths

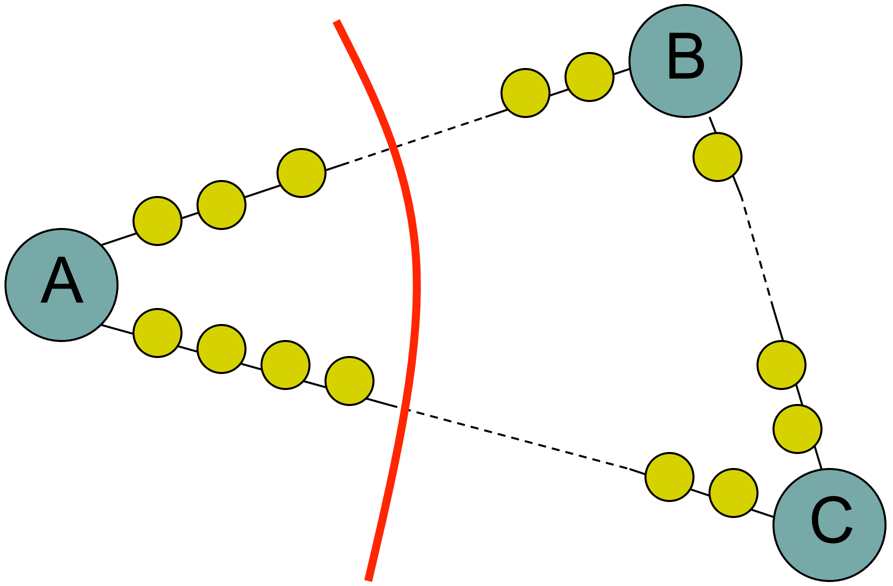


Shortest paths



Shortest paths

Nothing will change as we expand the frontier until we've gone out 100 levels



Ideas?

Dijkstra’s algorithm

```
map<int,int> shortest paths(int start,
                const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

Uses a priority queue to keep track of the next shortest path from the starting vertex

Vertices are kept in three sets:

- “***visited***”: those vertices who’s correct paths have been found. This occurs when a vertex is popped off the queue
- “***frontier***”: those vertices that we know about and have **a** path for, but not necessarily the vertices’ shortest paths. Vertices on the frontier are in the queue
- “***rest***”: the remaining vertices that we have not seen yet

Dijkstra's algorithm

```
map<int,int> shortest paths(int start,
                 const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

BFS

```
enqueue start;
while (queue not empty) {
   dequeue v;

   if (v is not visited) {
     visit v;
     enqueue all of v's neighbors;
   }
}
```

Dijkstra’s algorithm

```
map<int,int> shortest paths(int start,
                 const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
         ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

BFS

```
enqueue start;
while (queue not empty) {
   dequeue v;

   if (v is not visited) {
     visit v;
     enqueue all of v’s neighbors;
   }
}
```

- “parents” keeps track of shortest path
- only keep track of what the next vertex on the shortest path is

Dijkstra's algorithm

```
map<int,int> shortest paths(int start,
                const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
         ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

BFS

```
enqueue start;
while (queue not empty) {
  dequeue v;

  if (v is not visited) {
    visit v;
    enqueue all of v's neighbors;
  }
}
```

Dijkstra's algorithm

```
map<int,int> shortest paths(int start,
                 const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

BFS

```
enqueue start;
while (queue not empty) {
  dequeue v;

  if (v is not visited) {
    visit v;
    enqueue all of v's neighbors;
  }
}
```

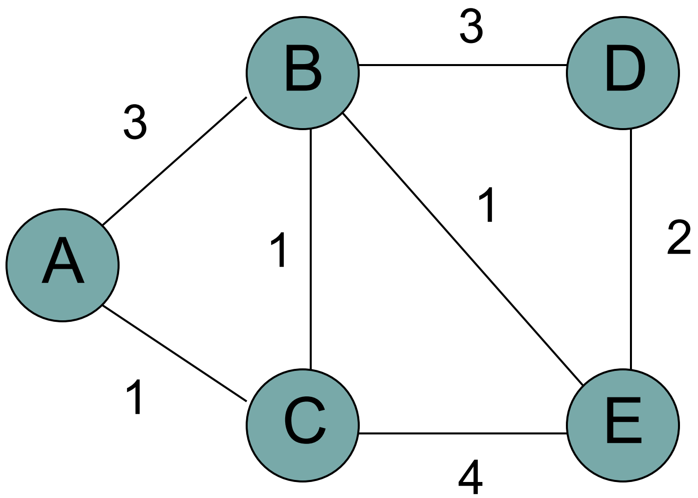
Dijkstra's algorithm

```
map<int,int> shortest paths(int start,
              const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
  } // end while
  return parents;
}
```

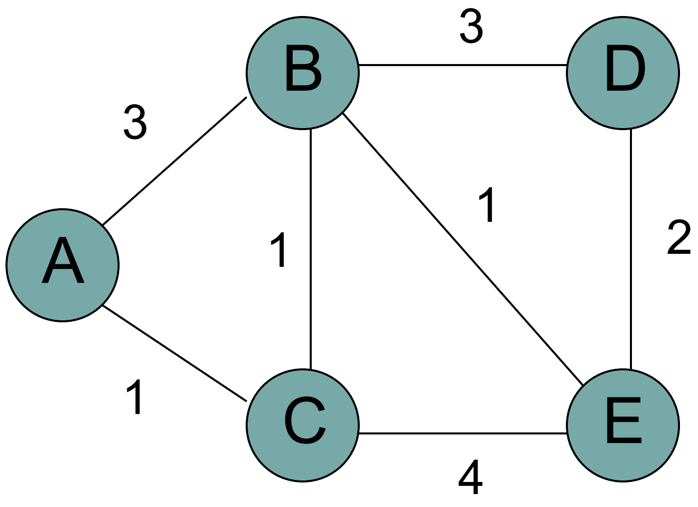
BFS

```
enqueue start;
while (queue not empty) {
  dequeue v;

  if (v is not visited) {
    visit v;
    enqueue all of v's neighbors;
  }
}
```

```
map<int,int> shortest paths(int start,
                 const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
         ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```



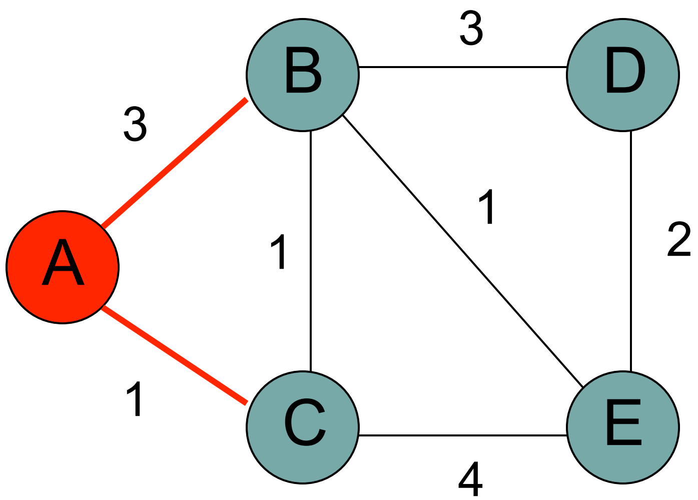
Heap	Parent
A 0	A: A



```
map<int,int> shortest paths(int start,
              const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

Heap	Parent
	A: A

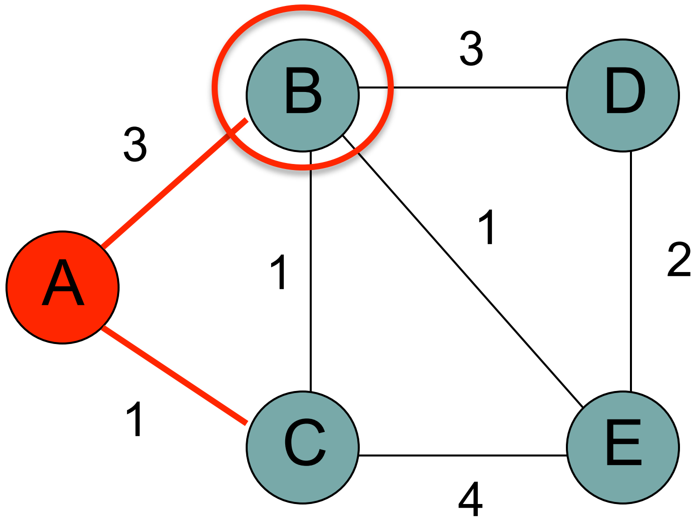
```
map<int,int> shortest paths(int start,
              const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
  } // end while
  return parents;
}
```



Heap

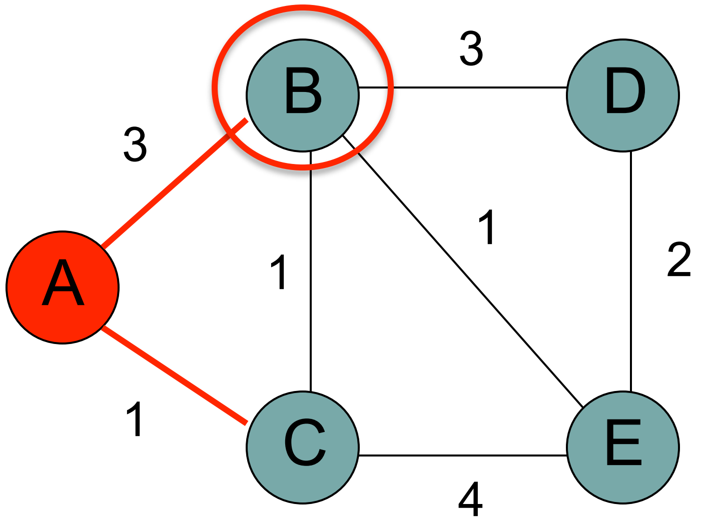
Parent

A: A



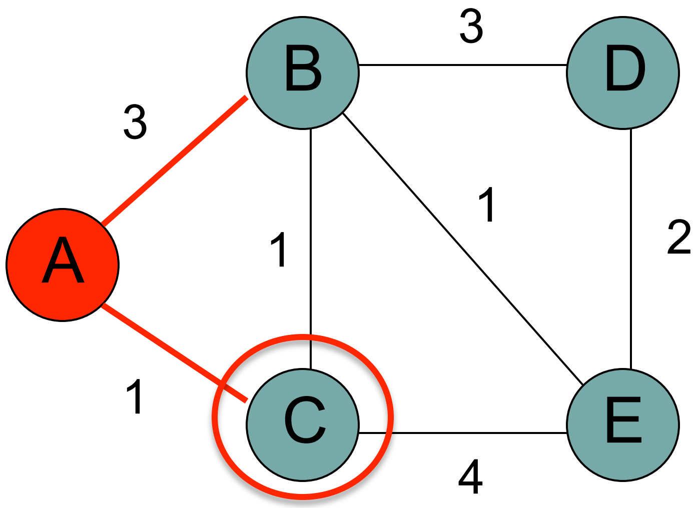
```
map<int,int> shortest paths(int start,
                 const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
         ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

Heap	Parent
B 3	A: A
	B: A



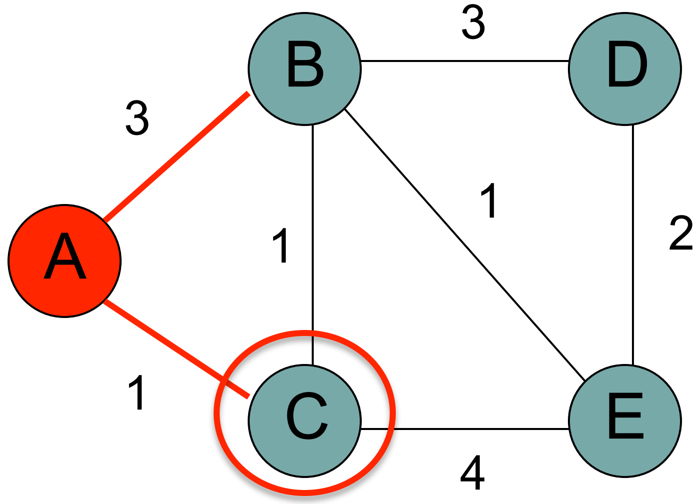
```
map<int,int> shortest paths(int start,
                const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
         ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

Heap	Parent
B 3	A: A
	B: A



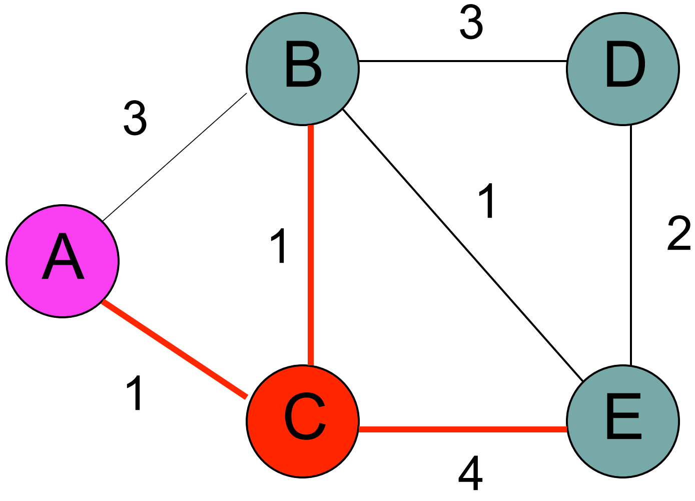
```
map<int,int> shortest paths(int start,
                const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
         ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
  } // end while
  return parents;
}
```

Heap	Parent
C 1	A: A
B 3	B: A
	C: A



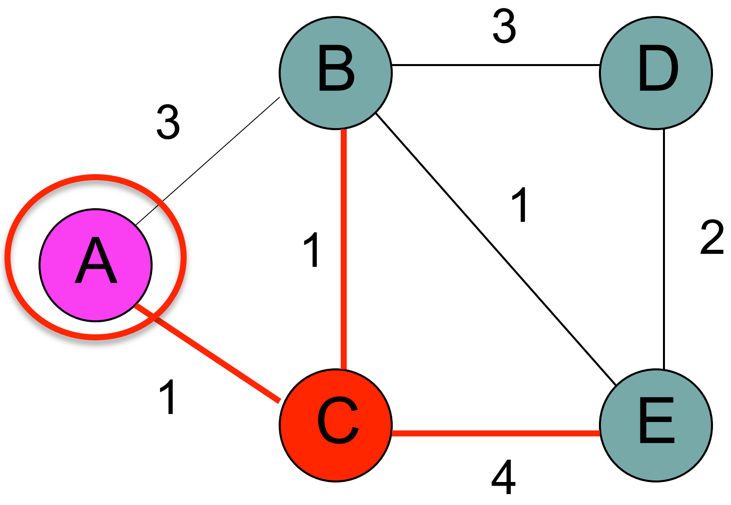
```
map<int,int> shortest paths(int start,
                const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

Heap	Parent
B 3	A: A
	B: A
	C: A



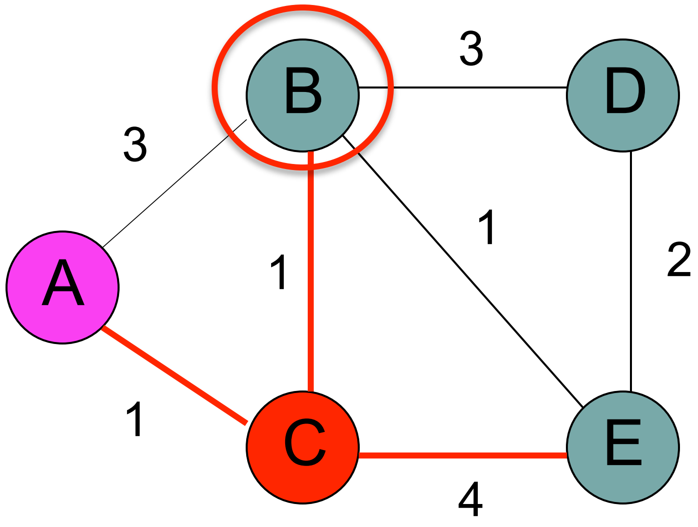
```
map<int,int> shortest paths(int start,
            const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
  } // end while
  return parents;
}
```

Heap	Parent
B 3	A: A
	B: A
	C: A



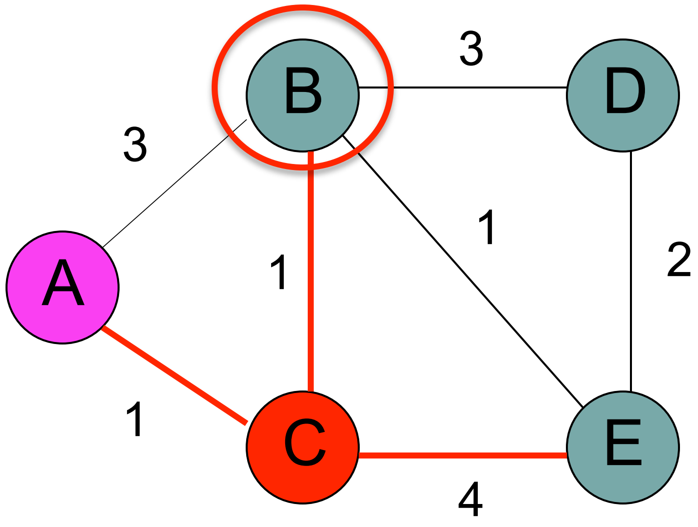
```
map<int,int> shortest paths(int start,
            const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
  } // end while
  return parents;
}
```

Heap	Parent
B 3	A: A
	B: A
	C: A



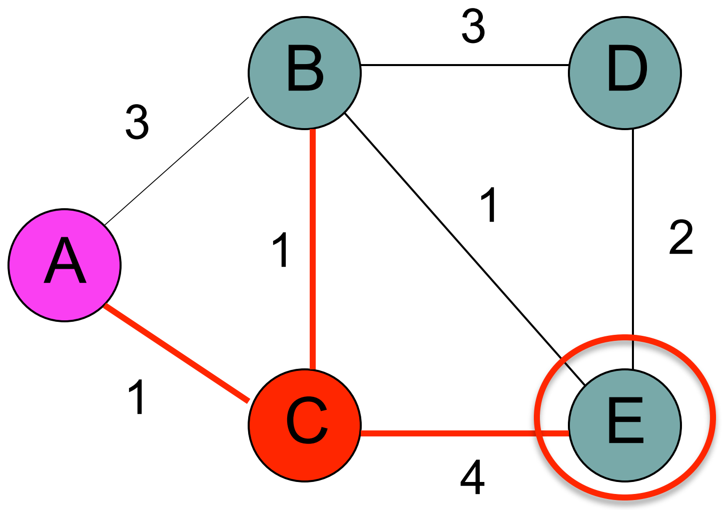
```
map<int,int> shortest paths(int start,
                const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
         ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

Heap	Parent
B 2	A: A
	B: C
	C: A



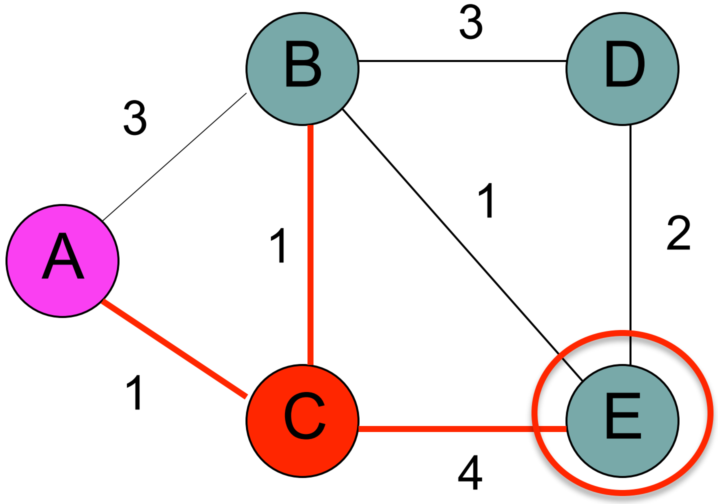
```
map<int,int> shortest paths(int start,
                 const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

Heap	Parent
B 2	A: A
	B: C
	C: A



```
map<int,int> shortest paths(int start,
              const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

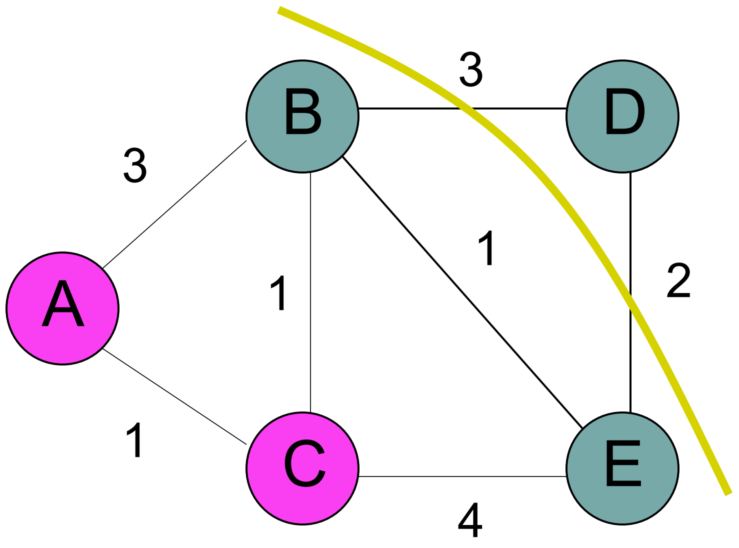
Heap	Parent
B 2	A: A
E 5	B: C
	C: A
	E: C



```
map<int,int> shortest paths(int start,
            const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
  } // end while
  return parents;
}
```

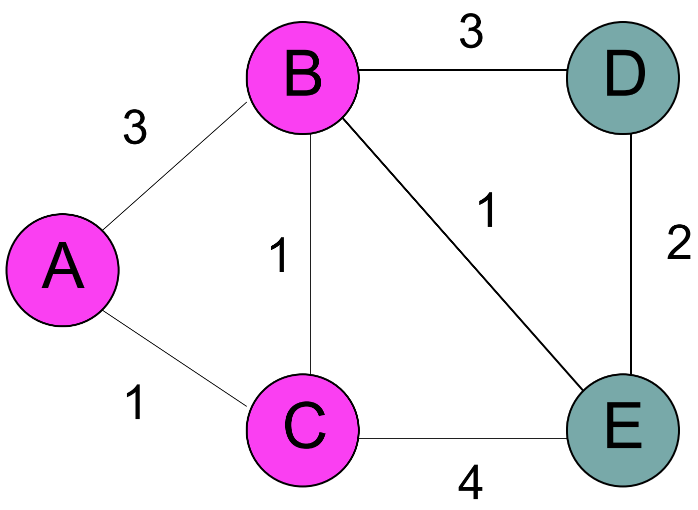
Heap	Parent
B 2	A: A
E 5	B: C
	C: A
	E: C

Frontier: all nodes reachable from starting node within a given distance



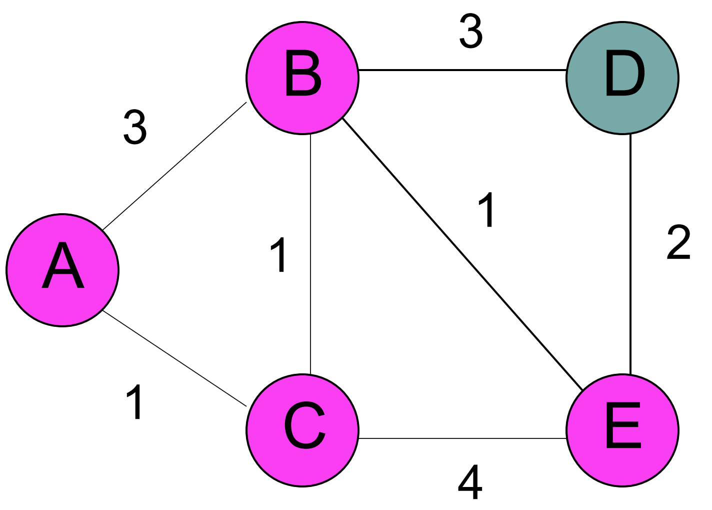
```
map<int,int> shortest paths(int start,
                 const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
         ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

Heap	Parent
E 3	A: A
D 5	B: C
	C: A
	D: B
	E: B



```
map<int,int> shortest paths(int start,
            const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
  } // end while
  return parents;
}
```

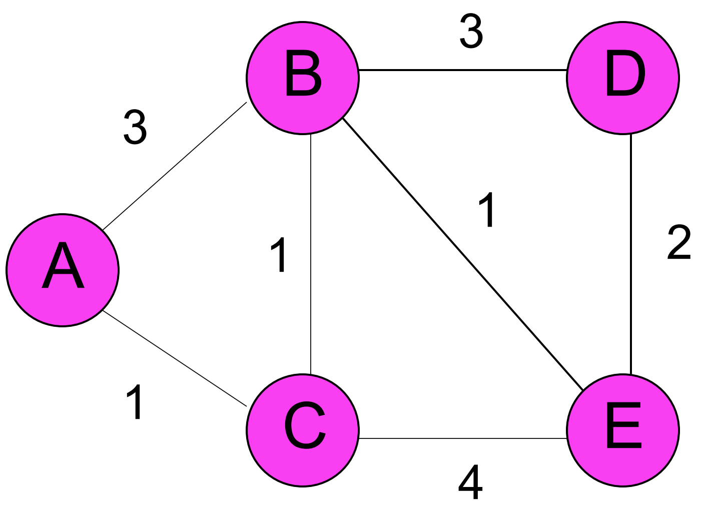
Heap	Parent
D 5	A: A
	B: C
	C: A
	D: B
	E: B



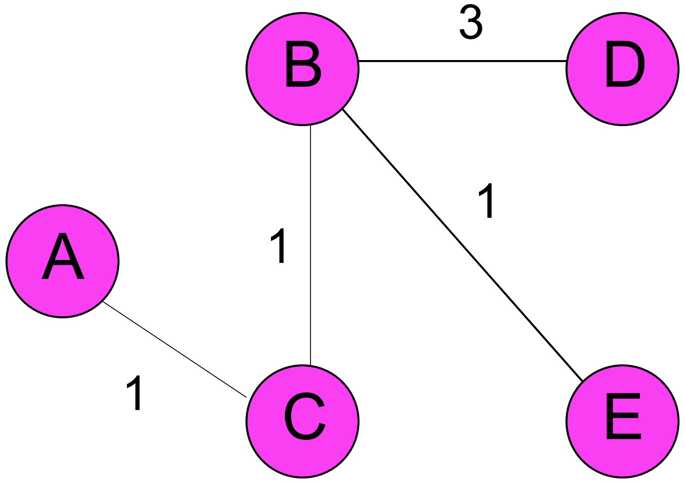
```
map<int,int> shortest paths(int start,
                const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
         ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
    } // end while
  return parents;
}
```

Heap	Parent
	A: A
	B: C
	C: A
	D: B
	E: B

```
map<int,int> shortest paths(int start,
                const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
  } // end while
  return parents;
}
```



Heap	Parent
	A: A
	B: C
	C: A
	D: B
	E: B



```
map<int,int> shortest paths(int start,
                const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {
    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited){
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }
  } // end while
  return parents;
}
```